Optimizing Hardware and Software for Gaussian Elimination

Computer Systems Engineering (EDA332)

DST18_49

Algot Johansson
Eric Guldbrand

May 15, 2018

1 Introduction

This project aims to create a fast and cost efficient hardware-software solution in the MIPS processor architecture for transforming a matrix to upper triangular form using Gaussian elimination. This is done by writing optimized MIPS assembler code and selecting a well-functioning memory/cache setup.

The final product is evaluated based on overall efficiency, measured as price $\times$ performance where price is a value in Chalmers-dollar (C$) and performance is the time in microseconds it took to transform the matrix.

2 Software Development

The first step was to create a functional program without regard to it's performance, except for making sure to use pointers rather than index calculations for terminating loops. This was done by naively implementing a basic algorithm (see figure 1 in PM). After first implementation the code was cleaned up and register usage was reorganized, then several improvements were made to increase performance.

All calculations for the last row were removed and replaced by a loop that statically fills it with zeroes and a one at the end. This could be done since that row will always look the same and none of it's values are actually used by the algorithm. In how the loops were constructed this required moving out pivot row element divisions for the second-to-last row outside the main loop, but this had insignificant performance impact.

Floating point stalls were reduced by replacing N divisions by 1 division and N multiplications. This was done because for each pivot element, one loop will divide several elements with the same denominator. Since division is significantly slower than multiplication, it could favourably be replaced with a multiplication of one over the denominator.

The biggest single improvement was obtained by using double load/store instead of float load store. Double and float load/store both take 1 cycle to complete, but the former loads 2 words into 2 registers, and the latter only 1. This change was implemented by performing two multiplications and two subtractions in each iteration instead of just one and then increasing the element pointer by two words per iteration instead of one. The tricky bit was to align the double load/store to an even word in memory, this meant having to add 2 words every other loop instead of 1 word every loop. This was solved using a XOR operation to vary an offset between 1 and 0 words to allow jumping two words to the right every other row. This meant that one element too much (the left-most) was calculated every other row. But it was overwritten anyway so it did not change the result.

Where two registers had different values but increased by the same amount each iteration of a loop one of them could sometimes be removed and the other simply use an offset to represent the one removed.

Finally, instructions were moved around to fill load-use time wherever possible. This sometimes included decreasing initial pointers by one word, adding to it in the middle of the loop and using offsets to make sure the same word would be accessed throughout each iteration.

An important note about developing the software is that care had to be taken to make sure the memory was not accessed unnecessarily. For instance, one failed optimization was to bake the last static ”zero loop” into the main one to remove the overhead of an extra loop. This failed because doing so meant caching a larger part of memory which in turn lead to more data cache stalls.

3 Hardware Setup

In the final setup, the instruction cache is direct-mapped and 128 byte (32 words) in size. This small cache with no set association is the best choice since the main loop is only 31 instructions long and can thus be cached in its entirety. Keeping the main loop short is the main reason why no loops were unrolled. Through testing, it was found that 4 blocks with a block size of 8 words was the fastest configuration for this cache. The block replacement policy is LRU (although FIFO would most likely have given the same performance).

The data cache is 2-way set-associative and 256 bytes (64 words) in size. This was the best since there is seldom more than two blocks fighting for the same address and thus it's not worth the cost of getting it. This was the smallest cache that could contain two rows ($24 \cdot 2 = 48$ words) of data at the same time. Because of the LRU write policy a smaller cache would unload the entire pivot-element row during every iteration of the innermost loop which would cause a large number of data cache misses. The write policy was write back.

Since the data memory is accessed so often the best memory was found to be type 2, the expensive but faster option. The best write buffer was 4 words (1 block) large. This buffer was tightly followed in performance by that of 8 words. The latter resulted in many fewer write buffer stalls but just not enough to outweigh its cost.

4 Tests of different configurations

Table 1 displays the result of multiple tests of the same software using varying memory/-cache setups. Each test varies only one variable. Largely, each new test uses the previously best test as a baseline and varies one variable from that. These tests does not cover every possible setup and there could be variations of 2 or more variables that in conjunction could create a better result. This is however unlikely since the preliminary tests used to find a reasonable hardware setup during development varied multiple variables and quickly found the setup these final tests improve upon and verify.

In order, the tested variables are: write buffer size, memory access time, D-cache block size, D-cache size (varying block amount), D-cache set size, I-cache block size, I-cache size (varying block amount), I-cache set size.

The final product efficiency is calculated using the formula

$$\frac{c \cdot p}{f} \tag{1}$$

Table 1: Test results of different memory configurations. The following variables are tested, in order: write buffer size, memory access time, D-cache block size, D-cache size (varying block amount), D-cache set size, I-cache block size, I-cache size (varying block amount), I-cache set size.

I-Cache			**D-Cache**			**Memory**			**Performance**	
Set Size *Blocks*	**Blocks**	**Block Size** *Words*	**Set Size** *Blocks*	**Blocks**	**Block Size** *Words*	**Access Time** *Cycles*		**Write Buffer** *Words*	**Cycle Count**	**μsC$**
1	8	4	2	16	4	14	3	12	71407	782
1	8	4	2	16	4	14	3	8	71537	555
1	8	4	2	16	4	14	3	4	73979	554
1	8	4	2	16	4	14	3	0	97523	704
1	8	4	2	16	4	22	4	4	91621	584
1	8	4	2	8	8	14	3	4	89280	669
1	8	4	2	32	2	14	3	4	84442	632
1	8	4	2	8	4	14	3	4	98947	650
1	8	4	2	32	4	14	3	4	68408	583
1	8	4	1	16	4	14	3	4	92814	658
1	8	4	4	16	4	13	3	4	89852	712
1	4	8	2	16	4	14	3	4	73925	554
1	2	16	2	16	4	14	3	4	76394	572
1	16	4	2	16	4	14	3	4	73979	595
2	8	4	2	16	4	14	3	4	73979	554

where c is the number of clock cycles taken for program execution, f is the frequency in MHz and p is the total price of hardware components. To calculate p, refer to the project assignment PM. The result of this formula is a number with the unit μsC$, representing the efficiency of the product, with less being better. In the goals for this project, an efficiency of at least 900 was to be attained, with one of 650 or better being set as a stretch goal.

From table 1, one of the best configuration obtained has an efficiency score of 554 μsC$ and uses the following hardware setup:

- Instruction Cache - Direct mapped, 4 blocks, 8 words per block, LRU.
- Data Cache - 2-way set associative, 16 blocks, 4 words per block, LRU, Write Back.
- Memory - Type 2.
- Write Buffer - 4 words.

In this configuration the CPU will cost 2 C$. The instruction cache will cost 0 C$ extra, the data cache will cost 0.25 C$ extra and slow down the clock speed to 450 MHz, The write buffer will cost $4 * 0.03$ C$ extra and the memory will cost 1 C$ extra.

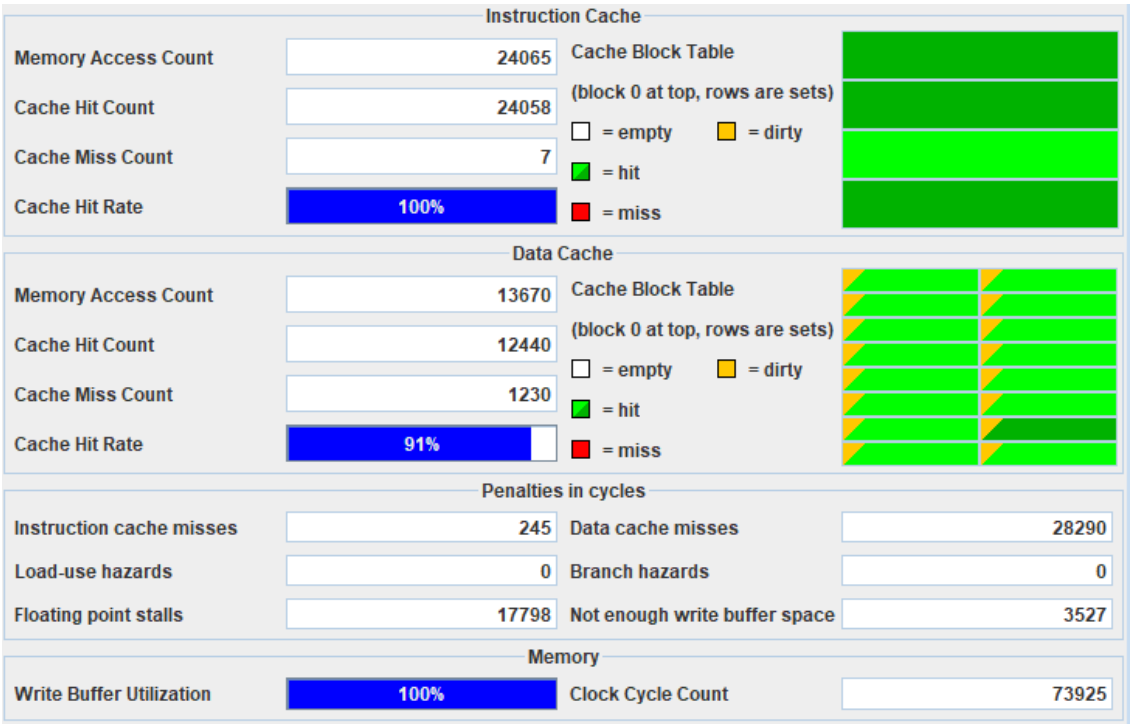

Figure 1: The result of the run with one of the best set of parameters.

The full MARS performance measurement of the best result can be seen in figure 1. There are very few cache misses on the instruction cache while data cache constitutes the single largest source of cycle consumption.